FlightLinux Project

Embedded Testbed
Technical Report

December 10, 2000

Updated Jan. 23, 2002

Patrick H. Stakem
QSS Group, Inc.

Revision History

December 10, 2000	Initial Release
Sept 2001	Included information on selected UoSat-12 code development
Oct. 2001	Included information on debugging
Dec. 2001	Rewrote and edited for deliverable.

Introduction

The intent of this technical report is to define the testbed facilities required to develop and test the FlightLinux software product. Based on the processor architectures defined in the Target Architecture Report, this Embedded Testbed Technical Report will define what is required to develop code for those architectures. This report also will define the readiness of the various testbeds. A specific architecture. UoSat-12 from SSTL with a custom 80386EX-processor board, has been defined for the FlightLinux flight experiment. The associated testbed facilities are defined in an appendix at the end of this report. A most important part of the process is the "make" file for the flight load, and this, although tedious, is included in an appendix as well.

This report focuses on facilities for developing the code for FlightLinux and then loading and testing it with the proper processor architecture. A testbed is more than a processor architecture. The LAN architecture will be discussed in a separate report, and the bulk memory approach also warrants a separate report. Linux is written in the c language, and numerous cross-compilers are available, hosted on pc and other platforms. The open-source GNU tools are used to develop the FlightLinux code, hosted on a pc platform. These tools are multi-target and will support any of the hardware architectures likely to be used. In addition, the testbed may include certain interfaces to allow simulation of the various spacecraft systems.

The following section explains the role of this report in the larger context of the FlightLinux Project.

The purpose of this section is to define the steps to FlightLinux implementation and the reports that have been produced as milestones in the FlightLinux process in addition to explaining their interrelationships. The goal of the FlightLinux Project is an on-orbit flight demonstration and validation of the FlightLinux operating system.

FlightLinux is a customized copy of a standard Linux distribution, adapted to the unique environment of a spacecraft embedded control computer.

The first demonstration of FlightLinux in space will be on the 80386EX processor of the currently in-orbit UoSat-12 spacecraft of Surrey Space Technology, Ltd (UK). As a basis, we are currently using the ELKS (Embeddable Linux Kernel Subset) distribution due to its small size. We will migrate to BlueCat Linux from LynuxWorks and add real-time features as required. Custom device drivers will address the unique aspects of the spacecraft architecture.

We have defined the steps to a space-flight demonstration of the Linux operating system. Regardless of the implementation architecture, certain pivotal issues must be defined. This will be done in a series of reports. These reference reports will be archived together in one place along with ongoing research related to the topics. The key issues include: the architecture of the target systems, the nature of application software, the architecture

of an onboard LAN, and the requirements for support, the architecture of the onboard storage system, the requirements for support, and the nature and design of the software development testbed.

The Target Architecture Technical Report examines the current, near-term, and projected computer architectures that will be used on board spacecraft. The resulting list allows examination of the feasibility and availability of Linux. The choice of the actual architecture for implementation will be determined more by opportunity of a flight than by choice of the easiest or most optimum architecture.

The *POSIX Report* examines and documents the POSIX-compliant aspects of Linux and other Flight Operating systems as well as the POSIX-compliant nature of legacy flight application software. This is an ongoing effort by GSFC Code 582, the Flight Software Branch.

The Onboard LAN Architecture Report discusses: 1) the physical level interfaces on existing and emerging missions and 2) the device drivers required to support IP over these interfaces. Ongoing work in this area is being done by the CCSDS committee and the OMNI Project (GSFC, Code 588). The choice of a demonstration flight will define which interfaces will need to be implemented first. In addition, those interfaces with COTS drivers, and those for which device drivers need to be defined will be delineated.

The Bulk Memory Device Driver Report will define the approach to be taken to implement the Linux file system in the bulk memory ("tape recorder") of the spacecraft onboard computer. It will define which elements are COTS and which need to be developed.

These reports are living documents and will be updated to document new developments. The reports will be stand-alone, but will reference the other reports as required. A major purpose of the reports will be to collect in one area the COTS aspects of the specific aspect of the FlightLinux implementation so that attention may be focused on the remaining "missing pieces."

Background

This work was conducted under task NAS5-99124-297, with funding by the NASA Advanced Information Systems Technology (AIST) Program, NRA-99-OES-08. The work is conducted by personnel of QSS Group, Inc. in partnership with NASA/GSFC Code 586 (Science Data Systems), Code 582 (Flight Software), and Code 588 (Advanced Architectures and Automation).

The FlightLinux project has the stated goal of providing an on-orbit flight demonstration of the software within the contract period. Numerous other Linux efforts exist within the GSFC flight software community. For example, the Triana flight code is currently being ported to Linux and most "legacy" flight code is being examined and modified for Posix compliance.

Management Summary

Code development, test, and integration environments for all of the architectures defined in the *Target Architecture Report* are currently in-place and available, or defined. The initial implementation of FlightLinux will be on the Intel 80386EX architecture of the UoSat-12 embedded computer, a custom board from Surrey Space Technology, Ltd.

The following table, updated from the one in the *Target Architecture Report*, identifies the most likely candidates for FlightLinux implementation in the near term.

Table 1. Candidate for FlightLinux Near-Term Implementation

	Flight Computer Linux Port Feasibility	
target	Base architecture	Assessment
WIRE - SMEX SCS	i80386/80387	COTS: Tune for limited memory resources
RAD6000	R/6000 - Power PC	COTS
RH32	MIPS R3000	COTS
Mongoose-V	MIPS R3000	COTS: Need to modify for lack of memory mgt
RHPPC	Power PC	COTS
RAD750	Power PC	COTS
ERC32	SPARC	COTS
Sandia Pentium	Pentium	COTS
Ericsson Space	Thor	COTS RTEMS
Surrey UoSat	80386EX	COTS Need to modify 80386 version
SNAP-1	StrongARM	COTS with minor modifications

Based on this information, we determined the testbed and development environment required for these architectures. We also determined if the testbed and tools were already available to the FlightLinux Project, and what would need to be procured to provide the necessary capability. As the focus of activity has been on the 80386EX architecture of the UoSat-12, it will be the major focus of this report.

Lessons Learned

1. Deploy development and debugging tools as early as possible. This allows the development and test team to get familiar with the tools and develop a toolbox of applications.

2. Remote access to the test system saves time. The test system can be arbitrarily located. This saves considerable time during the development and debug cycle.

3. Don't re-invent the wheel. With the Linux system, almost anything you can imagine already exists as code on the web.

Facilities

The facilities available to the FlightLinux Project include a hardware laboratory at QSS Group, Inc. (7404 Executive Blvd, Suite 400, Lanham, MD 20706 - approximately 1 mile from GSFC), the GSFC Flight Software Laboratory and the GSFC Science Data Systems Laboratory. The following table shows the development and testbed environment for the various targets.

Target	Code Development and Testbed Environment
i80386/80387	Breadboard at QSS Lab
80386EX	Breadboard, at QSS Lab; SSTL breadboard at GSFC OMNI
LabRAD6000	PowerPC at GSFC SDS Lab
RH32	Windows-NT-based cross-compiler at QSS Lab
Mongoose	Breadboard at GSCF Flight Software Lab RHPPC PowerPC at GSFC SDS lab
RAD750	GSFC Flight Software Lab
ERC32	Sun Workstation at QSS Lab
Pentium	Breadboard at QSS Lab
Thor	GSFC Flight Software Lab
StrongArm	Purchased unit. Vendor identified. Approx, cost $2k

Most of the likely architectures are covered by equipment and facilities that are in-place and available. We procured the RedHat Embedded Development Kit (EDK), which allows us to configure the software load from a PC platform to an embedded platform. For the 80386EX Processor, preliminary work can be done on any basic 80386 architecture, but a 80386EX-specific card is best for testing.

Differences between a 80386DX, 80386SX, and the 80386EX

We must first differentiate between design parameters dictated by the CPU and those set by the board design. Also, there is no requirement for the embedded board to be "PC-compatible", although this does simplify the use of standard software. In fact, the designers of the SSTL flight board kept the architecture close to that of a PC. This allows the use of COTS device drivers for critical interfaces (such as the task clock)and I/O locations.

The 80386 Processor is a circa-1985 32-bit CPU from Intel Corporation. The 80386sx Processor is an 80386 with a 16-bit external bus. The 80386EX is the core of an 80386 with additional interfaces for embedded application. The 386's popularity peaked around 1991. Since then, it has waned to the point that it is virtually non-existent on the market today.

386 Processors (from "Upgrading & Repairing PCs, Eighth Edition" (ref 11)

"The 386 can execute the real-mode instructions of an 8086 or 8088, but in fewer clock cycles. The 386 was as efficient as the 286 in executing instructions, which means that

the average instruction takes about 4.5 clock cycles. Probably the most important feature of this chip is its available modes of operation, which are Real Mode, Protected Mode, and Virtual 86 Mode. Standard Linux operates in Protected Mode. Intel extended the memory-addressing capabilities of the 386 Protected Mode with a memory management unit (MMU) that provides advanced memory paging and program switching.

Real Mode on a 386 chip is an 8086-compatible mode. In Real Mode, the 386 essentially is a much faster "turbo PC" limited to 1 megabyte of conventional memory, just like systems based on the 8086 chip. DOS and any software written to run under DOS require this mode to run. A minimalist version of Linux called ELKS can runs in this mode, the mode in which the Surrey loader software leaves the processor.

Numerous variations of the 386 chip exist. What follows is a brief discussion of: the baseline 80386DX, the 80386SX used in one of the breadboards, and the 80386EX variant used for the flight article.

386DX Processors

The 386DX chip was the first of the 386-family members that Intel introduced. The 386 is a full 32-bit processor with 32-bit internal registers, a 32-bit internal data bus, and a 32-bit external data bus. The 386 contains 275,000 transistors in a Very Large Scale Integration (VLSI) (circuit. The chip comes in a 132-pin package and draws approximately 400 milliamperes (ma). The 386 has a small power requirement, because it is made of Complementary Metal Oxide Semiconductor (CMOS) (materials. The CMOS design enables devices to consume extremely low levels of power. The 386DX can address 4G of physical memory. Its built-in virtual memory manager enables software designed to take advantage of enormous amounts of memory to act as though a system has 64 Terabytes of memory

386SX Processors

The 386SX was designed for system designers who were looking for 386 capabilities at 286-system prices. Like the 286, the 386SX was restricted to 16 bits when communicating with other system components such as memory. Internally, the 386SX is identical to the DX chip; the 386SX has 32-bit internal registers and can, therefore, run 32-bit software. The 386SX uses a 24-bit memory-addressing scheme like that of the 286, rather than the full 32-bit memory address bus of the standard 386. The 386SX, therefore, can address a maximum 16M of physical memory rather than the 4G of physical memory that the 386DX can address. Before it was discontinued, the 386SX was available in clock speeds ranging from 16 to 40MHz.

80386EX Processors

The 80386EX model includes the memory management features of the baseline 80386, along with an interrupt controller, a watchdog timer, sync/async serial I/O, DMA control, parallel I/O and dynamic memory refresh control. These devices are DOS-compatible in

the sense that their I/O addresses, dma and interrupt assignments correspond with an IBM PC board-level architecture. The DMA controller is, however, an enhanced superset of the 8237A DMA controller. The 80386EX processor core is static.

The 80386EX includes two DMA channels, three channels of 8254 timer/counter, dual 8259A interrupt controller functionality, a full-duplex synchronous serial I/O channel, two channels of 8250A asynchronous serial I/O, watchdog timer, 24 lines of parallel I/O, and support for dram refresh. The 80386EX can interface with the 80387 math co-processor, and the SSTL breadboard is equipped with an 80387SL.

Specific Device Drivers

A large number of device drives for custom I/O interfaces are available in open-source form for Linux. The physical-level I/O interfaces most likely to be required include asynchronous serial, synchronous serial, 1553/1773, CAN (ISO 11898), and ethernet (IEEE 802.3 10-base-T). The device driver needs to be customized for the specific physical layer hardware used to implement the interface. We have identified existing software drivers for all of these. The UoSat-12 OBC uses async serial (debug only) synchronous serial, 10Base-T, and Controller Area Network (CAN).

FlightLinux-specific Kernel Enhancements or Application Code

Certain extensions to a standard Linux kernel, beyond an embedded version, will have to be made for the unique space flight environment. These enhancements may take the form of application code, running under the kernel; this issue is currently "to be decided." Such enhancements will include a bulk memory device driver/file system (using bulk memory as a file system - to be discussed in a subsequent report), a memory scrub (wash) routine to periodically check and correct memory for radiation-induced errors, and a watchdog timer reset routine, to detect radiation-induced latchup of the processor.

Non-BIOS Systems

The UoSat-12 80386EX embedded board does not contain a BIOS, the firmware-based basic I/O system that is common in desktop computers. The functions that a BIOS provides include power-on self test, hardware configuration establishment, and software environment establishment. The UoSat-12 board does have a 32-kilobyte ROM that contains the Loader code. Two versions are available: one for loading from the async serial maintenance port and another for loading via the CAN bus. The loader code includes an Initialization routine, which provides some of the functionality of the BIOS in the sense of configuring the hardware and software environment. Run time support to programs is not provided. After the code is loaded, the ROM is mapped out of the memory space.

The Linux system in general, at least the 80x86-based implementations, rely on a BIOS for the initial system load. When a BIOS is not present, as in an embedded system, the functionality must be provided by other means. Once the Linux kernel is up and running,

Linux does not rely on the BIOS. Architectures other than the 80x86 contain firmware that may be broadly classified as a "BIOS."

Embedded Debugging Tools

Because the embedded system does not have the usual human interfaces such as keyboard and screen, alternative approaches must be found for debugging. One of these is the gnu debugger (gdb). When the target system (in this case, the SSTL 80386EX board) does not have a console, the gdb will run remotely on the host, with several minimal modules in the target and a link via the serial port. We are implementing the gdb on the UoSat-12 breadboard. Other UoSat-12 Breadboard debugging options include:

1. BlueCat Vendor Support - We started with the idea of using the BlueCat release of Linux, and we have copies of Version 2 and 3. We are currently using the ELKS distribution of Linux, because it is very simplified, and can be brought up, for example in real mode on the 80386; it does not require protected mode, with its associated complexities. The idea was to get something working with ELKS, then switch to BlueCat. One approach is to upgrade to the latest (version 4) release of BlueCat Linux, and get the associated priority support option. This would make available to us a BlueCat application engineer. The cost of this option is being evaluated. The Omni lab also currently uses BlueCat, but does not have a copy of Version 4.

2. ICE - an in-circuit emulator, uses a pod to replace the cpu, that cables to a box. A 80386EX ICE unit may be available at GSFC for loan. They are very expensive pieces of equipment. This turns out not to be an option with the UoSat-12 breadboard, as the cpu chip is soldered in. The UoSat unit is more of a proto-flight unit than a breadboard.

3. Logic Analyzer - There is a logic analyzer available in the Omni Lab. This unit can be considered a generic ICE - it can capture and display logic signals, can be triggered by predefined events, and can buffer a series of states on logic line. It would normally be used on the cpu's address and data bus, and some selected control signals. It is not processor-specific. There is a learning curve in its use, and the operator must understand software-hardware interactions.

4. Rom-based Monitor. On the UoSat-12 board, the ROM based monitor contains the load and dump code, and rudimentary hardware set-up routines. It is NOT a BIOS. We have the ability to plug a ROM emulator into the ROM socket, or to program custom proms. However, this may be a misleading approach, because we cannot, of course, change the rom contents on the flight unit.

5. A rom emulator plugs into the rom's socket, replacing the rom. This is also sometimes referred to as a Prom-ICE. Since the prom is replaced by ram on the debugging system, many options are possible. This may prove to be viable for the UoSat.

Bibliography

The following two technical reports produced under the FlightLinux contract are available on the Project website: http://FlightLinux.gsfc.nasa.gov

1. *POSIX Flight Software Report,* FlightLinux Project, Sept. 25, 2001.

2. *Target Architecture Report,* FlightLinux Project, December 15, 2001.

References on Embedded Cross-Development Environments:

4. Red Hat Linux Site: http://www.redhat.com/support/manuals/gnupro99r1/1_GS/int01.html#Embedded_cross-configuration_support

5. Cole, Bernard, "Remote Embedded Debugging in a Connected World," *Embedded.com,* Aug. 22, 2001; http://www.embedded.com/story/OEG20010822S0042

6. Cole, Bernard, "Debug and Test in Distributed Systems," *Embedded.com*, Sept 24, 2001; http://www.embedded.com/story/OEG20010924S0111

7. *The Future of Embedded Systems Debugging*, 1999, http://hmr.dasan.co.kr/papers/whitepapers/FutureOfESD/index.html

8. *gdb, the gnu debugger*. http://redhat.com/devnet/articles/embedgdb.htmlalso, http://luv.asn.au/overheads/prog/debugging.html

9. "Basics of Embedded Debugging," *Applied Microsystems,* http://www.amc.com/techcenter/whitepapers/basics/basics-1.html

10. Biederman, Eric, "About Linux BIOS," *Linux Journal*, December 2001.

11. Mueller, Scott. *Upgrading and Repairing PCs, Eighth Edition*, Que Books.

12. Short, Kenneth L. Embedded Microprocessor System Design, Prentice Hall, 1998, ISBN 0-13-249467-1.

Appendix A. Development Facilities for the UoSat-12 FlightLinux Flight Experiment.

This section describes the specific development facilities for the UoSat-12 software. These facilities consist of a number of Linux-based Intel boxes located at QSSMEDS for software development, and a breadboard facility for initial code testing. In addition, a breadboard facility acquired from SSTL represents the best model of the actual onboard architecture. This breadboard resides at Goddard Space Flight Center (GSFC) and is accessed from QSSMEDS remotely.

Table A-1.Differences Between Breadboard Units

Parameter	QSS Breadboard	SSTL Breadboard	Flight Unit
Processor	80386sx	80386ex	80386ex
Speed	20, 25, 33,40 Mhz	8, 16, 25 Mhz	8, 16, 25 Mhz
Coprocessor	none	80387sl	80387sl
Main memory	512k-4 megabyte	4 megabyte	4 megabyte
Extended mem	none	32 megabyte	32 megabyte
Async	2	2	not connected
Sync	none	4 full duplex	4 full duplex
Lan	10base-T	10base-T	10base-T
CAN bus	none	yes	yes
Software load	floppy	firmware loader	ViaCAN link, using firmware

The UoSat-12 breadboard and loading facility acquired by and administered by the OMNI Project, GSFC Code 588. This facility consists of a breadboard, flight-like UoSat-12 OBC, a 28-volt bench power supply, a Windows NT-based computer, and associated cables.

NT-Computer Host

This unit is a standard Intel-based desktop PC unit running the Windows-NT operating system and hosting both the "PCAnywhere" software and the SSTL loader program. It is connected to the GSFC LAN and to the OBC breadboard via a asynchronous serial port and a CAN interface with a SSTL-supplied PCI CAN card.

OBC386 Hardware

The OBC is based on Intel's 386EX 32-bit microcontroller. This microcontroller consists of an industry-standard 386SX microprocessor core and a number of peripherals. The OBC supports a maximum of 4 megabytes of program storage, 128 megabytes of Solid State Data Recorder (referred to as Ramdisk in this document) memory, and 32 kilobytes of firmware storage. The program memory is protected by a majority voting system. Two separate communication controllers handle the serial communication with the up and

downlinks. On-board communication is supported by a medium-speed 10BASE-2 Ethernet controller. Telecommand and telemetry is handled by an on-board Controller Area Network (CAN) controller. This controller allows other bus nodes to issue "Reset" commands, switch I/O multiplexers, and enable/disable the Ramdisk without application software support.

386EX Microcontroller

The OBC386 is based on Intel's 386SX-compatible microcontroller. This micro-controller was chosen due to its enhanced microprocessor core, build-in peripherals, and extended addressing capabilities of up to 64 Mbytes. It also provides a software-compatible upgrade path from SSTL's primary OBC186 (based on Intel 80C186). The 386EX is available in commercial and extended temperature range; a military screened 883B version is not available at the time of writing this documentation.

Procedure for Assembling a Test Program

This section describes a procedure to assemble, link, and locate a test program written in Assembly language for the OBC386. The assembler and linker are both Microsoft products whereas the locater is developed by CSI. The batch file assembles the file (command line argument) using the Microsoft assembler**(** MASM) without linking it. The next line links the file using the OBC386 debugger library (OBC386D.LIB). This library contains several routines to read and write to the universal asynchronous receiver-transmitter **(**UART). The OBC386.LIB file (without the "D" extension) implements the same routines but writes to ISCC channel 0 instead. After linking, the file is located to address 0x500 by the CSILOC program. The file can then be uploaded and executed using the WINLOAD utility.

Batch File

```
@echo off
echo OBC386 Assemble, Link and Locate
echo Output file is D.BIN @ 000500 physical
ml /W3 /c %1.asm
link %1.obj, d.exp,d.map,OBC386D.LIB /MAP;
csiloc d.cxd
del %1.bin
del *.obj
copy d.bin %1.bin
```

"AL.BAT" file

```
// OBC386 CSI Locate file
exec d.exp
binary // Create binary file
CPU 386 // Target is 386
locate _TEXT :: 0500p // Locate at 500 physical
```

Bench Test Setup

For uploading new software, the UoSat-12 engineering model uses one of the 386EX built-in UART's (TX/RX UART0). This UART is connected to PL21 pin 1 and 2. The second UART is connected to pin 1 and 2 of jumper OPTI2. Both are connected to the host box. The associated Window-NT-based loader software runs under Windows NT.

Software Development Tools

The OBC software is written in two languages: "8086 assembler" and "c." Tools for both are required. For the assembler, two very different variations are found. One is the standard "Intel format" and the other is the AT&T format. These require different assemblers, and use very different syntax.

Under the Linux operating system, assemblers and the gnu c compiler (gcc) are readily available.

Remote Operation of the Development Facility

The Windows-NT machine hosting the SSTL loader software is co-located with the breadboard and is interconnected with both asynchronous serial and can busses. The NT box is connected to the GSFC infrastructure LAN and then to the Internet. This equipment is located in the OMNI Laboratory.

Software development is done at QSSMEDS on various desktop Linux machines. A load image is built and stored as a disk file. This is transferred to the NT loader machine by means of the "PC-Anywhere" software. This software allows complete console control of the NT system, while maintaining acceptable security. The NT machine may even be rebooted under control of the pc-anywhere. It implements sufficient layers of security and protection to allow remote access over the GSFC LAN. Remote access to the testbed facility provides a level of efficiency, because 1) the development team and the breadboard need not be co-located and 2) the travel requirements are virtually eliminated.

Appendix B. - "Typical" Make File for a UoSat-12 Load

```
(cd kernel; make)
make[1]: Entering directory `/home/swozny/elks/kernel'
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o sched.o sched.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o printk.o printk.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o sleepwake.o sleepwake.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o dma.o dma.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o version.o version.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o sys.o sys.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o stubs.o stubs.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o fork.o fork.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o exit.o exit.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o time.o time.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o module.o module.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o signal.o signal.c
ar rcs kernel.a sched.o printk.o sleepwake.o dma.o version.o sys.o stubs.o fork.o exit.o
time.o module.o signal.o
make[1]: Leaving directory `/home/swozny/elks/kernel'
(cd fs; make)
make[1]: Entering directory `/home/swozny/elks/fs'
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o devices.o devices.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o fcntl.o fcntl.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o inode.o inode.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o file_table.o file_table.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o enamei.o enamei.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o ioctl.o ioctl.c
bcc -D__KERNEL__ -O -I../include  \
```

```
-0 -c -o open.o open.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o read_write.o read_write.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o exec.o exec.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o select.o select.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o pipe.o pipe.c
ar rcs fs.a devices.o fcntl.o inode.o file_table.o enamei.o ioctl.o open.o read_write.o
exec.o select.o pipe.o
make[1]: Leaving directory `/home/swozny/elks/fs'
(cd lib; make)
make[1]: Entering directory `/home/swozny/elks/lib'
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o chqueue.o chqueue.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o string.o string.c
ar rcs lib.a chqueue.o string.o
sync
make[1]: Leaving directory `/home/swozny/elks/lib'
(cd net; make)
make[1]: Entering directory `/home/swozny/elks/net'
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o socket.o socket.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o protocols.o protocols.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o ipv4/af_inet.o ipv4/af_inet.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o unix/af_unix.o unix/af_unix.c
bcc -D__KERNEL__ -O -I../include  \
-0 -c -o nano/af_nano.o nano/af_nano.c
ar rcs net.a socket.o protocols.o ipv4/af_inet.o unix/af_unix.o nano/af_nano.o
make[1]: Leaving directory `/home/swozny/elks/net'
bcc -D__KERNEL__ -O -i \
-0 -nostdinc -Iinclude -c -o init/main.o init/main.c
00465                                   *
00466                                   *       Create the stack frame
00467                                   *
00470                                   *
00471                                   * Get the high word
00472                                   *
00474                                   *
00475                                   *       Delay the higher word
00476                                   *
```

```
00481                   *
00482                   *       Delay a complete low word loop time
00483                   *
00487                   *
00488                   *       Now back around for the next high word
00489                   *
00492                   *
00493                   *       Delay for the low part of the time
00494                   *
00502                   *
00503                   *       Recover stack frame and return
00504                   *
00465                   *
00466                   *       Create the stack frame
00467                   *
00470                   *
00471                   *       Get the high word
00472                   *
00474                   *
00475                   *       Delay the higher word
00476                   *
00481                   *
00482                   *       Delay a complete low word loop time
00483                   *
00487                   *
00488                   *       Now back around for the next high word
00489                   *
00492                   *
00493                   *       Delay for the low part of the time
00494                   *
00502                   *
00503                   *       Recover stack frame and return
00504                   *
(cd arch/i86; make Image)
make[1]: Entering directory `/home/swozny/elks/arch/i86'
(cd tools; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/tools'
gcc -I /home/swozny/elks/include -o build build.c
make[2]: Leaving directory `/home/swozny/elks/arch/i86/tools'
gcc -E -traditional -I/home/swozny/elks/include/ -o boot/setup.s boot/setup.S
as86 -0 -o boot/setup.o boot/setup.s
ld86 -0 -s -o boot/setup -M boot/setup.o > Setup.map
(cd kernel; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/kernel'
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o strace.o strace.c
```

```
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o printreg.o printreg.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o system.o system.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o irq.o irq.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o irqtab.o irqtab.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o process.o process.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o bios16.o bios16.c
00012                           * Things we want to save - direction flag BP ES
00017                           * We have to save DS carefully.
00025                           * Load the register block from the table
00031                           * ES in 16
00034                           * Flags in 20
00036                           * Flags to end up with
00038                           * AX final
00040                           * Stack is now Flags, AX
00042                           * DS value final
00044                           * Load BX
00046                           * Stack now holds stuff to restore followed by the call
values
00047                           * for flags,AX
00048 ********** DS is now wrong we cannot load from the array again**********/
00049                           * DS desired
00051                           * AX desired
00053                           * Flags desired
00055                           *
00056                           *Do a disk interrupt.
00057                           *
00059                           *
00060                           *  Now recover the results
00061                           *
00062                           * Make some breathing room
00067                           * Stack is now returned FL, BX, AX, DS
00072                           * Recover our DS segment
00075 ********** We can now use the bios data table again ***************
00077                           * Save the old DS
00080                           * Save the old AX
00083                           * Save the old BX
00093                           * Pop the returned flags off
00095                           *
00096                           * Restore things we must save
00097                           *
```

```
00012                                   * Things we want to save - direction flag BP ES
00017                                   * We have to save DS carefully
00025                                   * Load the register block from the table
00031                                   * ES in 16
00034                                   * Flags in 20
00036                                   * Flags to end up with
00038                                   * AX final
00040                                   * Stack is now Flags, AX
00042                                   * DS value final
00044                                   * Load BX
00046                                   * Stack now holds stuff to restore followed by the call
values
00047                                   * for flags,AX
00048 DS is now wrong we cannot load from the array again**********/
00049                                   * DS desired
00051                                   * AX desired
00053                                   * Flags desired
00055                                   *
00056                                   *  Do a disk interrupt
00057                                   *
00059                                   *
00060                                   *  Now recover the results
00061                                   *
00062                                   * Make some breathing room
00067                                   * Stack is now returned FL, BX, AX, DS
00072                                   * Recover our DS segment
00075    We can now use the bios data table again ***************
00077                                   * Save the old DS
00080                                   * Save the old AX
00083                                   * Save the old BX
00093                                   * Pop the returned flags off
00095                                   *
00096                                   *  Restore things we must save
00097                                   *
sh mkentry.sh > tmp.o
mv tmp.o entry.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o entry.o entry.c
00002                                   *
00003                                   * The call table - autogenerated from syscall.dat
00004                                   *
00082                                   *
00083                                   * Despatch a syscall (called from syscall_int)
00084                                   * Entry: ax=function code, stack contains parameters
00085                                   *
00096                                   *
```

```
00097                                   * Unimplemented calls
00098                                   *
00002                                   *
00003                                   * The call table - autogenerated from syscall.dat
00004                                   *
00082                                   *
00083                                   *  Despatch a syscall (called from syscall_int)
00084                                   *  Entry: ax=function code, stack contains parameters
00085                                   *
00096                                   *
00097                                   *  Unimplemented calls
00098                                   *
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o signal.o signal.c
bcc -Wall -D__KERNEL__ -O -I../../../include \
-0 -c -o timer.o timer.c
ar rcs akernel.a strace.o printreg.o system.o irq.o irqtab.o process.o bios16.o entry.o
signal.o timer.o
make[2]: Leaving directory `/home/swozny/elks/arch/i86/kernel'
(cd lib; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/lib'
bcc -0 -c -0 -c -o idiv.o idiv.s
00001                                   | idiv.s
00002                                   | idiv_ doesn't preserve dx (returns remainder in it)
00001                                   | idiv.s
00002                                   | idiv_ doesn't preserve dx (returns remainder in it)
bcc -0 -c -0 -c -o idivu.o idivu.s
00001                                   | idivu.s
00002                                   | idiv_u doesn't preserve dx (returns remainder in it)
00001                                   | idivu.s
00002                                   | idiv_u doesn't preserve dx (returns remainder in it)
bcc -0 -c -0 -c -o imod.o imod.s
bcc -0 -c -0 -c -o imodu.o imodu.s
00001                                   | imodu.s
00002                                   | imodu doesn't preserve dx (returns quotient in it)
00012                                   | instruction queue full so xchg slower
00001                                   | imodu.s
00002                                   | imodu doesn't preserve dx (returns quotient in it)
00012                                   | instruction queue full so xchg slower
bcc -0 -c -0 -c -o imul.o imul.s
00001                                   | imul.s
00002                                   | imul_, imul_u don't preserve dx
00001                                   | imul.s
00002                                   | imul_, imul_u don't preserve dx
bcc -0 -c -0 -c -o isl.o isl.s
00001                                   | isl.s
```

```
00002                                   | isl, islu don't preserve cl
00001                                   | isl.s
00002                                   | isl, islu don't preserve cl
bcc -0 -c -0 -c -o isr.o isr.s
00001                                   | isr.s
00002                                   | isr doesn't preserve cl
00001                                   | isr.s
00002                                   | isr doesn't preserve cl
bcc -0 -c -0 -c -o isru.o isru.s
00001                                   | isru.s
00002                                   | isru doesn't preserve cl
00001                                   | isru.s
00002                                   | isru doesn't preserve cl
bcc -0 -c -0 -c -o inport.o inport.s
00001                                   | int inw( int port );
00002                                   | reads a word from the i/o port  port  and returns it
00015                                   | int inw_p( int port );
00016                                   | reads a word from the i/o port  port  and returns it
00001                                   | int inw( int port );
00002                                   | reads a word from the i/o port  port  and returns it
00015                                   | int inw_p( int port );
00016                                   | reads a word from the i/o port  port  and returns it
bcc -0 -c -0 -c -o inportb.o inportb.s
00001                                   | int inb( int port );
00002                                   | reads a byte from the i/o port  port  and returns it
00016                                   | int inb( int port );
00017                                   | reads a byte from the i/o port  port  and returns it. Uses
an in
00018                                   | from port 0x80 to slow the process down.
00001                                   | int inb( int port );
00002                                   | reads a byte from the i/o port  port  and returns it
00016                                   | int inb( int port );
00017                                   | reads a byte from the i/o port  port  and returns it. Uses
an in
00018                                   | from port 0x80 to slow the process down.
bcc -0 -c -0 -c -o outport.o outport.s
00001                                   | void outw( int value, int port );
00002                                   | writes the word  value  to  the i/o port  port
00015                                   | void outw_p( int value, int port );
00016                                   | writes the word  value  to  the i/o port  port
00001                                   | void outw( int value, int port );
00002                                   | writes the word  value  to  the i/o port  port
00015                                   | void outw_p( int value, int port );
00016                                   | writes the word  value  to  the i/o port  port
bcc -0 -c -0 -c -o outportb.o outportb.s
00001                                   | void outb( char value, int port);
```

```
00002                                   | writes the byte  value  to  the i/o port  port
00015                                   | void outb_p( char value, int port);
00016                                   | writes the byte  value  to  the i/o port  port
00001                                   | void outb( char value, int port);
00002                                   | writes the byte  value  to  the i/o port  port
00015                                   | void outb_p( char value, int port);
00016                                   | writes the byte  value  to  the i/o port  port
bcc -0 -c -0 -c -o peekb.o peekb.s
00001                                   | int peekb( unsigned segment, char *offset );
00002                                   | returns the (unsigned) byte at the far pointer
segment:offset
00001                                   | int peekb( unsigned segment, char *offset );
00002                                   | returns the (unsigned) byte at the far pointer
segment:offset
bcc -0 -c -0 -c -o peekw.o peekw.s
00001                                   | int peekw( unsigned segment, int *offset );
00002                                   | returns the word at the far pointer  segment:offset
00001                                   | int peekw( unsigned segment, int *offset );
00002                                   | returns the word at the far pointer  segment:offset
bcc -0 -c -0 -c -o peekd.o peekd.s
00001                                   | long peekd( unsigned segment, unsigned int *offset );
00002                                   | returns the dword at the far pointer  segment:offset
00001                                   | long peekd( unsigned segment, unsigned int *offset );
00002                                   | returns the dword at the far pointer  segment:offset
bcc -0 -c -0 -c -o pokeb.o pokeb.s
00001                                   | void pokeb( unsigned segment, char *offset, char value
);
00002                                   | writes the byte  value  at the far pointer  segment:offset
00001                                   | void pokeb( unsigned segment, char *offset, char value
);
00002                                   | writes the byte  value  at the far pointer  segment:offset
bcc -0 -c -0 -c -o pokew.o pokew.s
00001                                   | void pokew( unsigned segment, int *offset, int value );
00002                                   | writes the word value  at the far pointer  segment:offset
00001                                   | void pokew( unsigned segment, int *offset, int value );
00002                                   | writes the word value  at the far pointer  segment:offset
bcc -0 -c -0 -c -o poked.o poked.s
00001                                   | void pokew( unsigned segment, unsigned *offset,
unsigned long value );
00002                                   | writes the word value  at the far pointer  segment:offset
00001                                   | void pokew( unsigned segment, unsigned *offset,
unsigned long value );
00002                                   | writes the word value  at the far pointer  segment:offset
bcc -0   -D__KERNEL__ -I../../../include \
-0 -c -o bitops.o bitops.c
bcc -0   -D__KERNEL__ -I../../../include \
```

```
-0 -c -o memmove.o memmove.c
bcc -0 -c -0 -c -o string.o string.s
bcc -0 -c -0 -c -o fmemset.o fmemset.s
00002                           1 void farmemset(char *off, int seg, int value, size_t
count)
00002                           1 void farmemset(char *off, int seg, int value, size_t
count)
bcc -0 -c -0 -c -o border.o border.s
00001                           | border.s
00001                           | border.s
bcc -0 -c -0 -c -o laddl.o laddl.s
00001                           | laddl.s
00001                           | laddl.s
bcc -0 -c -0 -c -o landl.o landl.s
00001                           | landl.s
00001                           | landl.s
bcc -0 -c -0 -c -o lcmpl.o lcmpl.s
00001                           | lcmpl.s
00002                           | lcmpl, lcmpul don't preserve bx
00012                           | don't need to preserve bx
00021                           | b (below) becomes lt (less than) as well
00023                           | ge and already ae
00024                           | else make gt as well as a (above)
00026                           | clear ov and mi, set ne for greater than
00034                           | clear ov, set mi and ne for less than
00001                           | lcmpl.s
00002                           | lcmpl, lcmpul don't preserve bx
00012                           | don't need to preserve bx
00021                           | b (below) becomes lt (less than) as well
00023                           | ge and already ae
00024                           | else make gt as well as a (above)
00026                           | clear ov and mi, set ne for greater than
00034                           | clear ov, set mi and ne for less than
bcc -0 -c -0 -c -o lcoml.o lcoml.s
00001                           | lcoml.s
00001                           | lcoml.s
bcc -0 -c -0 -c -o ldecl.o ldecl.s
00001                           | ldecl.s
00001                           | ldecl.s
bcc -0 -c -0 -c -o ldivl.o ldivl.s
00001                           | ldivl.s
00002                           | bx:ax / 2(di):(di), quotient bx:ax, remainder di:cx, dx
not preserved
00013                           | bx:ax / di:cx, quot di:cx, rem bx:ax
00001                           | ldivl.s
```

```
00002                           | bx:ax / 2(di):(di), quotient bx:ax, remainder di:cx, dx
not preserved
00013                           | bx:ax / di:cx, quot di:cx, rem bx:ax
bcc -0 -c -0 -c -o ldivul.o ldivul.s
00001                           | ldivul.s
00002                           | unsigned bx:ax / 2(di):(di), quotient bx:ax,remainder
di:cx, dx not preserved
00013                           | unsigned bx:ax / di:cx, quot di:cx, rem bx:ax
00001                           | ldivul.s
00002                           | unsigned bx:ax / 2(di):(di), quotient bx:ax,remainder
di:cx, dx not preserved
00013                           | unsigned bx:ax / di:cx, quot di:cx, rem bx:ax
bcc -0 -c -0 -c -o leorl.o leorl.s
00001                           | leorl.s
00001                           | leorl.s
bcc -0 -c -0 -c -o lincl.o lincl.s
00001                           | lincl.s
00001                           | lincl.s
bcc -0 -c -0 -c -o lmodl.o lmodl.s
00001                           | lmodl.s
00002                           | bx:ax % 2(di):(di), remainder bx:ax, quotient di:cx, dx
not preserved
00014                           | bx:ax / di:cx, quot di:cx, rem bx:ax
00001                           | lmodl.s
00002                           | bx:ax % 2(di):(di), remainder bx:ax, quotient di:cx, dx
not preserved
00014                           | bx:ax / di:cx, quot di:cx, rem bx:ax
bcc -0 -c -0 -c -o lmodul.o lmodul.s
00001                           | lmodul.s
00002                           | unsigned bx:ax / 2(di):(di), remainder bx:ax,quotient
di:cx, dx not preserved
00013                           | unsigned bx:ax / di:cx, quot di:cx, rem bx:ax
00001                           | lmodul.s
00002                           | unsigned bx:ax / 2(di):(di), remainder bx:ax,quotient
di:cx, dx not preserved
00013                           | unsigned bx:ax / di:cx, quot di:cx, rem bx:ax
bcc -0 -c -0 -c -o lmull.o lmull.s
00001                           | lmull.s
00002                           | lmull, lmulul don't preserve cx, dx
00001                           | lmull.s
00002                           | lmull, lmulul don't preserve cx, dx
bcc -0 -c -0 -c -o lnegl.o lnegl.s
00001                           | lnegl.s
00001                           | lnegl.s
bcc -0 -c -0 -c -o lorl.o lorl.s
00001                           | lorl.s
```

```
00001                                   | lorl.s
bcc -0 -c -0 -c -o lsll.o lsll.s
00001                                   | lsll.s
00002                                   | lsll, lslul don't preserve cx
00001                                   | lsll.s
00002                                   | lsll, lslul don't preserve cx
bcc -0 -c -0 -c -o lsrl.o lsrl.s
00001                                   | lsrl.s
00002                                   | lsrl doesn't preserve cx
00024                                   | equivalent to +infinity in this context
00001                                   | lsrl.s
00002                                   | lsrl doesn't preserve cx
00024                                   | equivalent to +infinity in this context
bcc -0 -c -0 -c -o lsrul.o lsrul.s
00001                                   | lsrul.s
00002                                   | lsrul doesn't preserve cx
00001                                   | lsrul.s
00002                                   | lsrul doesn't preserve cx
bcc -0 -c -0 -c -o lsubl.o lsubl.s
00001                                   | lsubl.s
00001                                   | lsubl.s
bcc -0 -c -0 -c -o ltstl.o ltstl.s
00001                                   | ltstl.s
00002                                   | ltstl, ltstul don't preserve bx
00026                                   | clear ov and mi, set ne for greater than
00001                                   | ltstl.s
00002                                   | ltstl, ltstul don't preserve bx
00026                                   | clear ov and mi, set ne for greater than
gcc -I ../../../include -E -traditional -o setupw.s setupw.S
bcc -0 -c -0 -c -o setupw.o setupw.s
00002                                   | int setupw( unsigned segment, int *offset );
00003                                   | returns the word at the far pointer  0x9000:offset
00002                                   | int setupw( unsigned segment, int *offset );
00003                                   | returns the word at the far pointer  0x9000:offset
gcc -I ../../../include -E -traditional -o setupb.s setupb.S
bcc -0 -c -0 -c -o setupb.o setupb.s
00002                                   | int setupb(char *offset);
00003                                   | returns the (unsigned) byte at the far pointer
0x9000:offset
00002                                   | int setupb(char *offset);
00003                                   | returns the (unsigned) byte at the far pointer
0x9000:offset
bcc -0 -c -0 -c -o ldivmod.o ldivmod.s
00001                                   | ldivmod.s - 32 over 32 to 32 bit division and remainder
for 8086
```

```
00003                   | ldivmod( dividend bx:ax, divisor di:cx )  [ signed quot
di:cx, rem bx:ax ]
00004                   | ludivmod( dividend bx:ax, divisor di:cx ) [ unsigned
quot di:cx, rem bx:ax ]
00006                   | dx is not preserved
00009                   | NB negatives are handled correctly, unlike by the
processor
00010                   | divison by zero does not trap
00013                   | let dividend = a, divisor = b, quotient = q, remainder =
r
00014                   |       a = b * q + r  mod 2^32
00015                   | where:
00017                   | if b = 0, q = 0 and r = a
00019                   | otherwise, q and r are uniquely determined by the
requirements:
00020                   | r has the same sign as b and absolute value smaller than
that of b, i.e.
00021                   |       if b > 0, then 0 <= r < b
00022                   |       if b < 0, then 0 >= r > b
00023                   | (the absoulute value and its comparison depend on
signed/unsigned)
00025                   | the rule for the sign of r means that the quotient is
truncated towards
00026                   | negative infinity in the usual case of a positive divisor
00028                   | if the divisor is negative, the division is done by
negating a and b,
00029                   | doing the division, then negating q and r
00038                   | sign byte of b in dh
00040                   | sign byte of a in dl
00049                   | leave r = a positive
00060                   | both sign bytes 0
00066                   | remember b
00077                   | would overflow
00079                   | a in dx:ax, signs in bx
00082                   | q in di:cx, junk in ax
00084                   | signs in ax, junk in bx
00086                   | r in ax, signs back in dx
00088                   | r in bx:ax
00092                   | return q = 0 and r = a
00096                   | a initially minus, restore it
00100                   | remember sign bytes
00102                   | w in si:dx, initially b from di:cx
00105                   | q in di:cx, initially 0
00107                   | r in bx:ax, initially a
00108                   | use di:cx rather than dx:cx in order to
00109                   | have dx free for a byte pair later
```

```
00113                                   | finished if b > r
00117                                   | rotate w (= b) to greatest dyadic multiple of b <= r
00121                                   | w = 2*w
00124                                   | w was > r counting overflow (unsigned)
00126                                   | while w <= r (unsigned)
00131                                   | else exit with carry clear for rcr
00137                                   | q = 2*q
00140                                   | if w <= r
00147                                   | q++
00150                                   | r = r-w
00154                                   | w = w/2
00157                                   | while w >= b
00165                                   | sign bytes
00171                                   | else a initially minus, b plus
00173                                   | -a = b * q + r ==> a = b * (-q) + (-r)
00176                                   | use if r = 0
00178                                   | use a = b * (-1 - q) + (b - r)
00181                                   | q = -1 - q (same as complement)
00197                                   | (-a) = (-b) * q + r ==> a = b * q + (-r)
00199                                   | use if initial a was minus
00201                                   | a = (-b) * q + r ==> a = b * (-q) + r
00204                                   | use if r = 0
00206                                   | use a = b * (-1 - q) + (b + r) (b is now -b)
00001                                   | ldivmod.s - 32 over 32 to 32 bit division and remainder
for 8086
00003                                   | ldivmod( dividend bx:ax, divisor di:cx )  [ signed quot
di:cx, rem bx:ax ]
00004                                   | ludivmod( dividend bx:ax, divisor di:cx ) [ unsigned
quot di:cx, rem bx:ax ]
00006                                   | dx is not preserved
00009                                   | NB negatives are handled correctly, unlike by the
processor
00010                                   | divison by zero does not trap
00013                                   | let dividend = a, divisor = b, quotient = q, remainder =
r
00014                                   |       a = b * q + r  mod 2^32
00015                                   | where:
00017                                   | if b = 0, q = 0 and r = a
00019                                   | otherwise, q and r are uniquely determined by the
requirements:
00020                                   | r has the same sign as b and absolute value smaller than
that of b, i.e.
00021                                   |       if b > 0, then 0 <= r < b
00022                                   |       if b < 0, then 0 >= r > b
00023                                   | (the absoulute value and its comparison depend on
signed/unsigned)
```

```
00025                   | the rule for the sign of r means that the quotient is
truncated towards
00026                   | negative infinity in the usual case of a positive divisor
00028                   | if the divisor is negative, the division is done by
negating a and b,
00029                   | doing the division, then negating q and r
00038                   | sign byte of b in dh
00040                   | sign byte of a in dl
00049                   | leave r = a positive
00060                   | both sign bytes 0
00066                   | remember b
00077                   | would overflow
00079                   | a in dx:ax, signs in bx
00082                   | q in di:cx, junk in ax
00084                   | signs in ax, junk in bx
00086                   | r in ax, signs back in dx
00088                   | r in bx:ax
00092                   | return q = 0 and r = a
00096                   | a initially minus, restore it
00100                   | remember sign bytes
00102                   | w in si:dx, initially b from di:cx
00105                   | q in di:cx, initially 0
00107                   | r in bx:ax, initially a
00108                   | use di:cx rather than dx:cx in order to
00109                   | have dx free for a byte pair later
00113                   | finished if b > r
00117                   | rotate w (= b) to greatest dyadic multiple of b <= r
00121                   | w = 2*w
00124                   | w was > r counting overflow (unsigned)
00126                   | while w <= r (unsigned)
00131                   | else exit with carry clear for rcr
00137                   | q = 2*q
00140                   | if w <= r
00147                   | q++
00150                   | r = r-w
00154                   | w = w/2
00157                   | while w >= b
00165                   | sign bytes
00171                   | else a initially minus, b plus
00173                   | -a = b * q + r ==> a = b * (-q) + (-r)
00176                   | use if r = 0
00178                   | use a = b * (-1 - q) + (b - r)
00181                   | q = -1 - q (same as complement)
00197                   | (-a) = (-b) * q + r ==> a = b * q + (-r)
00199                   | use if initial a was minus
00201                   | a = (-b) * q + r ==> a = b * (-q) + r
```

```
00204                                   | use if r = 0
00206                                   | use a = b * (-1 - q) + (b + r) (b is now -b)
ar rcs lib86.a idiv.o idivu.o imod.o imodu.o imul.o isl.o isr.o isru.o inport.o inportb.o
outport.o outportb.o
peekb.o peekw.o peekd.o pokeb.o pokew.o poked.o bitops.o memmove.o string.o
fmemset.o border.o laddl.o
landl.o lcmpl.o lcoml.o ldecl.o ldivl.o ldivul.o leorl.o lincl.o lmodl.o lmodul.o lmull.o
lnegl.o lorl.o lsll.o lsrl.o lsrul.o
lsubl.o ltstl.o setupw.o setupb.o ldivmod.o
sync
make[2]: Leaving directory `/home/swozny/elks/arch/i86/lib'
(cd mm; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/mm'
bcc -D__KERNEL__ -O -Wall -I../../../include \
-0 -c -o init.o init.c
bcc -D__KERNEL__ -O -Wall -I../../../include \
-0 -c -o segment.o segment.c
bcc -D__KERNEL__ -O -Wall -I../../../include \
-0 -c -o malloc.o malloc.c
bcc -D__KERNEL__ -O -Wall -I../../../include \
-0 -c -o user.o user.c
ar rcs mm.a init.o segment.o malloc.o user.o
sync
make[2]: Leaving directory `/home/swozny/elks/arch/i86/mm'
(cd drivers/char; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/drivers/char'
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o bioscon.o bioscon.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o serial.o serial.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o lp.o lp.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o xt_key.o xt_key.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o init.o init.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o dircon.o dircon.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o ntty.o ntty.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o mem.o mem.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o meta.o meta.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o pty.o pty.c
bcc -0 -nostdinc -I../../../../include -D__KERNEL__ -O -0 -c -o bell.o bell.c
ar rcs chr_drv.a bioscon.o serial.o lp.o xt_key.o init.o dircon.o ntty.o mem.o meta.o pty.o
bell.o
sync
make[2]: Leaving directory `/home/swozny/elks/arch/i86/drivers/char'
(cd drivers/block; make)
make[2]: Entering directory `/home/swozny/elks/arch/i86/drivers/block'
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o genhd.o genhd.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o doshd.o doshd.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o ll_rw_blk.o ll_rw_blk.c
```

```
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o rd.o rd.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o floppy.o floppy.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o directhd.o directhd.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o init.o init.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -0 -c -o sibo_ssd.o sibo_ssd.c
bcc -0 -I../../../../include  -D__KERNEL__ -O -P -o ssd_asm.s ssd_asm.S
as86  -0 -o ssd_asm.o ssd_asm.s
ar rcs blk_drv.a genhd.o doshd.o ll_rw_blk.o rd.o floppy.o directhd.o init.o sibo_ssd.o
ssd_asm.o
make[2]: Leaving directory `/home/swozny/elks/arch/i86/drivers/block'
bcc -D__KERNEL__ -O -i \
2 -nostdinc -I/home/swozny/elks/include -c -o boot/crt1.o boot/crt1.c
gcc -E -traditional -I/home/swozny/elks/include/ -o boot/crt0.s boot/crt0.S
as86 -0 -0 -o boot/crt0.o boot/crt0.s
(cd ../.. ; ld86 -0 -i arch/i86/boot/crt0.o \
        arch/i86/boot/crt1.o \
        init/main.o \
        kernel/kernel.a fs/fs.a lib/lib.a net/net.a arch/i86/kernel/akernel.a arch/i86/lib/lib86.a
arch/i86/mm/mm.a \
         arch/i86/drivers/char/chr_drv.a arch/i86/drivers/block/blk_drv.a \
        -t -M -o arch/i86/boot/system > System.map)
tools/mkurlader  -c e000 64 Image e000 -a boot/setup e000 -s boot/system e060
  boot/setup: 108h Bytes (- a.out) @e000
  boot/system: 5cf8h Bytes (strip) @e060
--> Image: 62f8h Bytes @e000
!! No ROM-Signature at 00000
sync
make[1]: Leaving directory `/home/swozny/elks/arch/i86'
```